Winter 2019 - COMP 520 Compiler Design
Milestone 2 Report

Group 07
Zhuocheng Du - 260673297
Anqi Li - 260654876
Zisheng Wang - 260619653

1 Introduction

After milestone 1, our compiler can perform lexical and syntactic analysis for source programs written in GoLite programming language. In milestone 2, by implementing symbol tables and type-checking, our compiler can also perform semantic analysis on the source programs.

Symbol tables allows the compiler to keep track of defined variables and their respective types. It is implemented by searching for variable declarations while traversing the abstract syntax tree built for the source program. For every valid variable declaration, the identifier of the variable and its associated type are added to the symbol table of the scope where the declaration takes place. Scopes are created and then exited according to the scoping rules defined in Section 3. Every scope has its own unique symbol table. Moreover, when implementing symbol tables, we also check if variable uses are legal, which is done by checking if every identifier used can be found in its current scope or any of its parent scopes.

Type-checking allows the compiler to check if type compatibility is ensured in the source program. For every declaration, statement, or expression, whether it type-checks can be determined by verifying if respective rules defined in GoLite language specification[1] are satisfied.

2 Design Decisions

In this section, we explain how we implemented the non-trivial part of symbol tables and typechecking.

2. 1 Symbol Table

We implemented symbol tables as a cactus stack of hash tables[2]. Each hash tables contains its parent and a set of symbols. Symbols (struct SYMBOL) are defined as structs and contains a string of the name of the symbol, a Type (struct TYPE) of the symbol, a kind (enum symbolKind) to tell apart of different kinds of symbols e.g type declaration, variable declaration, function declaration, and constants.

2.1.1 Variable Declarations: Initialization

For each variable declaration, we will first put the symbol of the expression into the symbol table. If there was not an explicit type notation, the type of the expression will be assigned to the variable. If there was not an expression, there must be an explicit type notation and this type information will be assigned to the variable. When declaring a variable with a type of slice or array, if the slice's/array's element type is declared, we will add this new slice/array type into symbol table.

2.1.2 Types

We added BaseTypeKind, underLineType, and infer kind to our existing TYPE struct definition. BaseTypeKind is an enumeration which is used when the type is a base type (int, float64, bool, string, rune). UnderLineType is a TYPE pointer which points to the underlying type. This creates a list of Types. The head of such a list is the current type pointer, while the tail is one of the base type pointers. For arrays and slices, its underLineType is the type of inner components. For example, the underLineType of a []int is int, the underLineType of a [][]int is []int. For structs, the id of its underLineType is always "struct" since we will not need to resolve to struct's inner components. We also changed how we set the identifier for slices and array types: the identifier of slices is "[]" plus the identifier of its underLineType, while the identifier of array is "[size]" plus the identifier of its underLineType. For instance, the identifier of a type for an int array with size being 5 is "[5]int". Additionally, at symbol table phase, if a variable does not have an explicit type notation, we will assign a type of kind k_infer and will assign the correct type information later.
When initializing the top level symbol table, we added all base types. To put a new type declaration symbol into symbol table, we first check if the name of the type is neither main nor init, and then we check if the type is recursive, and if so, we check if it is valid recursively. After all checks have been passed, we will only put the new type into symbol table if its identifier is not a blank identifier. We also set up the base type kind information for later use.

For struct type declarations, we will open a new scope for its field declarations. The identifier and type of each field declaration is added to the new symbol table.

We created a helper function to determine if two types are equal: bool checkSameType(TYPE *t1, TYPE *t2, bool checkBaseType). t1 and t2 are the two types to be checked. If checkBaseType is true, we check if two types' underlying base types are the same. Two types are equal if they have the same kind. For slice kind, two types are equal if they have the same element type. For array kind, two types are equal if they have the same element type and have the same size. For structs, two types are equal if all their field declarations have the same types and are declared in the same exact order, and the identifier of the structs are either identical or absent. We also check (in a separate function) the declaration scope of types. Two types are not equal if they are declared in different scopes.

2.1.3 Assign Statements

There are two checks for assign statements. First, the program validates that the left-hand side of the assign statement can be resolved to something that is resolvable (i.e. The lowermost layer of it must be referenceable by an identifier). Next, we check that both sides of the assign statement share the same type. During our development, we realized that two types sharing the same identifier could be considered as different types if they were declared in different scopes, and those types shall not be compatible in an assign statement or a binary expression. We therefore redesigned the procedure of filling our symbol table in a way that each unique type is pointed to by only one pointer.

2.1.4 Short Variable Declarations

As long as there is one undeclared variable on the left-hand side, a short variable declaration is valid. Since our identifier list implementation assumed that all elements of the list will share the same type, we made a few helper functions , such as resolveType() and strToType(), to ensure that only one internal copy of each unique type is stored in our table. Please note that a blank identifier cannot be the only undeclared variable.

2.1.5 Switch Statements

When a switch statement is detected, we first create a new symbol table A to store the new variable declared before we go into the switch cases. The symbol table of the scope where the switch statement as a whole occurs is the parent of symbol table A. Another symbol table, i.e. table B, needs to be created to store any possible new variable declarations in the switch case clauses. Since all the variables declared before the case clauses can be reached inside the case clauses, we must set table A to be the parent of table B. In addition, we must verify that any variables used in switch cases have been declared before.

2.1.6 For Loops

Our compiler merged the procedure of creating new scoped table for all block statement (statements enclosed in curly brackets) other than those found in switch, a function, or a struct. A new scope is created whenever a for statement, an if statement, or an explicit is detected. For example, for the for-loop, we first create a new symbol table A to store the new variable declared in the first part of the for clause. The symbol table of the scope where the for statement as a whole occurs is the parent of symbol table A. Another symbol table, i.e. table B, needs to be created to store any possible new variable declarations in the body of the for statement. Since all the variables declared before the for body can be reached inside the for body, we must set table A to be the parent of table B.

2.1.7 Functions

A new scope is created for afunction. The function's name and its parameters are declared at the beginning of this new table. Although our parser supported parameterized return variable, we abandoned it due to the complexity of implementation.

Since our symbol table has a one-pass implementation, in order to print the function arguments, we have created a global variable where the string representation of the function's signature is temporarily stored.

2.2 Type-checking

We mainly followed the language specification to implement this part. Tricky cases are explained below.

2.2.1 Function Typechecking and Return Statements

If a function is declared to return void, a return statement is not necessary, but if a return statement ever occurs in its body, we check that it returns a NULL expression.[3]

If a function is declared to return non-void type, first we must check that it always returns (i.e. It always terminates). This is implemented in weeder. We traverse the function body, and for each statement encountered, we check if it is a terminating statement. Repeat so until there are no more statements in the function body. When we reach the last statement of the function body, if it is not a terminating statement, then we can determine that this function does not terminate and an error must be thrown.

Terminating statements are defined as below:

- A return statement is a terminating statement
- A for statement which is equivalent to an infinite loop is a terminating statement
- An if statement is a terminating statement if its body terminates and its associated else statement is a terminating statement.
- An else statement that does not contain an if statement is a terminating statement if its body terminates
- An else statement that contains an if statement is a terminating statement if the contained if statement terminates
- A switch statement is a terminating statement if it contains a default case and every case ends with a terminating statement

After we make sure that a non-void function always returns, we check whether it returns what it claims to return when declared. We verify this by passing the return type of the function into statement type-checking function. In this way, when a return statement is detected while traversing the function body, we can check if the returned expression has the same type as the passed-in function return type.

3 Scoping Rules

Any identifiers that have not been declared in a new scope may be used for variable declaration. They will shadow earlier declarations. Special cases include the parameter definition in a function declaration and the field definition in a struct declaration. In those cases, although declared types' names may be used as variables, their definitions as type will not be shadowed.

A new scope will be created in the following cases:

- In the body of for loop, if-else statements, or any other explicit block statement
- In the short statement of switch and the body of switch, please refer to Section 2.1.5
- In a function declaration, including its parameters
- In a struct declaration

A scope will be poped from our stack in the following cases:

- When a closing curly brace is encountered (so that the block terminates).

4 Test Cases

We designed our test cases by first reading the language specification and then deriving invalid cases from each section of the specification. The test cases submitted are only a subset of all test cases that we used to test our compiler. In order to fully test the functionality, apart from those created from scratch by ourselves (i.e. the submitted ones), we have also enriched our test set by taking test cases from various sources, including past GoLite project repositories[3] and the official GoLang repository[4].

5 Important Changes since Milestone 1

After running our submitted code for milestone 1 with the grading test cases, we found out the defects in our previous implementation and made the following changes accordingly:

1. We removed "&" from the list of valid unary operators
2. We excluded ' and \ from the set of characters that are valid for rune
3. We excluded 0x from the set of tokens that are valid for hex-based integer
4. We added checks of blank identifier for the right-hand side of assignment statements
5. We updated TYPE struct definition. Please refer to Section 2.1.2.

6 Work Distribution

Team members cooperated with one another by using Visual Studio Code with LiveShare extension so that everyone can view, edit, and execute the code simultaneously. Each team member has contributed to all parts of the milestone, including design and implementation of symbol table and type checking, and writing milestone report.

7 References

[1] Krolik, A. (2019). Milestone 2 Specifications. [online] Available at: https://www.cs.mcgill.ca/~cs520/2019/project/Milestone2_Specifications.pdf.
[2] Krolik, A. (2019). Course slides 7-symbol.pdf. [online] cs.mcgill.ca. Available at: https://www.cs.mcgill.ca/~cs520/2019/slides/7-symbol.pdf.
[3] Li, E. (2019). EmolLi/GoLite. [online] GitHub. Available at: https://github.com/EmolLi/GoLite.
[4] Golang/go. (2019). GitHub. [online] Available at: https://github.com/golang/go.